A task-based parallelism and vectorized approach to 3D Method of Characteristics (MOC) reactor simulation for high performance computing architectures

A task-based parallelism and vectorized approach to 3D Method of Characteristics (MOC) reactor simulation for high performance computing architectures

John R. Tramm[a,b,*], Geoffrey Gunow[b], Tim He[a], Kord S. Smith[b], Benoit Forget[b], Andrew R. Siegel[a]

[a]*Argonne National Laboratory, Center for Exascale Simulation of Advanced Reactors 9700 S Cass Ave, Argonne, IL 60439*

[b]*Massachusetts Institute of Technology, Department of Nuclear Science & Engineering 77 Massachusetts Avenue, 24-107, Cambridge, MA 02139*

Abstract

In this study we present and analyze a formulation of the 3D Method of Characteristics (MOC) technique applied to the simulation of full core nuclear reactors. Key features of the algorithm include a task-based parallelism model that allows independent MOC tracks to be assigned to threads dynamically, ensuring load balancing, and a wide vectorizable inner loop that takes advantage of modern SIMD computer architectures. The algorithm is implemented in a set of highly optimized proxy applications in order to investigate its performance characteristics on CPU, GPU, and Intel Xeon Phi architectures. Speed, power, and hardware cost efficiencies are compared. Additionally, performance bottlenecks are identified for each architecture in order to determine the prospects for continued scalability of the algorithm on next generation HPC architectures.

Keywords:
Method of Characteristics, Neutron Transport, Reactor Simulation, High Performance Computing

[*]Corresponding author. Tel.: +1 (847) 421-1534.

Email addresses: `jtramm@mcs.anl.gov` (John R. Tramm), `geogunow@mit.edu` (Geoffrey Gunow), `shuohe@mcs.anl.gov` (Tim He), `kord@mit.edu` (Kord S. Smith), `bforget@mit.edu` (Benoit Forget), `siegela@mcs.anl.gov` (Andrew R. Siegel)

1. Introduction

A central goal in computational nuclear engineering is the high fidelity simulation of a full nuclear reactor core. Full core simulations can potentially reduce design and construction costs, increase reactor performance and safety, and reduce the amount of nuclear waste generated. To date, however, the time to solution for a full core high-fidelity deterministic 3D calculation has rendered such calculations infeasible, even using leadership class supercomputers. However, with High Performance Computing (HPC) architectures evolving rapidly towards exascale class supercomputers, the computational horsepower required to accomplish full 3D deterministic reactor simulations may soon be available. One simulation approach, the 3D Method of Characteristics (MOC) technique, has the potential for fast and efficient performance on a variety of next generation HPC systems. While 2D MOC has long been used in reactor design and engineering as a medium-fidelity simulation method for smaller problems, the transition to 3D has only begun recently, and to our knowledge no 3D production codes have been developed [1, 2, 3].

Next generation supercomputer designs, such as GPUs and the Intel Phi, rely on a number of novel accelerator architectures with characteristics distinct from traditional CPUs [4]. This increased diversity of computational architectures highlights the need to gain a deep understanding of performance before committing to a specific implementation strategy, parallel programming language, target architecture, or parallelization scheme. Investment in a full scale 3D MOC production code must be done with confidence that the end product code will perform efficiently on a variety of exascale computing architectures. This imperative leads us to develop a set of "mini-apps", specifically *SimpleMOC* (previously presented by Gunow et al.[5]) and *SimpleMOC-kernel*, which mimic the computational performance of a full 3D MOC solver while abstracting away elements necessary to achieve full solutions – allowing for much easier performance analysis on a wide variety of hardware and architecture simulators.

In the present work, we present a brief overview of our formulation of the 3D MOC algorithm, as implemented in our mini-apps, and a discussion of how the algorithm is constructed to optimally map onto a variety of HPC architectures. Then, we compare performance data for CPU, GPU, and Intel Phi architectures, and analyze performance bottlenecks on each architecture. Power profiles of our mini-apps on several architectures are also presented in order to analyze power efficiency. In addition, performance is normalized against node hardware cost data to determine the cost efficiency of multiple architectures. Finally, we use this data to estimate the likelihood of continued scalability for the 3D MOC simulation algorithm on next generation HPC and exascale architectures.

1.1. Reactor Simulation via the Neutron Transport Equation

The goal of the simulation of a nuclear reactor is to answer several questions about the reactor in order to both make revisions to its initial design and to better predict its operational behavior. Two of the most important items to understand are: 1) the eigenvalue, or criticality, of the reactor, and 2) its power distribution. The eigenvalue determines the ratio of neutron populations between successive generations within the reactor. An eigenvalue of 1.0 indicates that the reactor is "critical" and therefore capable of operating at steady state in a controllable fashion. The power distribution within the reactor is also important to understand, as this governs the thermal-hydraulic design considerations as well as the speed at which the nuclear fuel burns up. n

The eigenvalue and power distribution are computed by numerically estimating the solution of the Boltzmann Equation, represented for a steady-state system as:

$$\begin{aligned}\vec{\Omega}\cdot\nabla\psi(\vec{r},\vec{\Omega},E)+\Sigma_t(\vec{r},E)\psi(\vec{r},\vec{\Omega},E) = \\ +\int_0^\infty dE'\int_{4\pi} d\Omega'\Sigma_s(\vec{r},\vec{\Omega}'\to\vec{\Omega},E'\to E)\psi(\vec{r},\vec{\Omega}',E') \\ +\frac{\chi(\vec{r},E)}{4\pi k_{eff}}\int_0^\infty dE'\nu\Sigma_f(\vec{r},E')\int_{4\pi} d\vec{\Omega}'\psi(\vec{r},\vec{\Omega}',E')\end{aligned} \quad (1)$$

Equation 1 defines the neutron transport equation where ψ is the angular neutron flux, $\vec{\Omega}$ is the angular direction vector, $\vec{r}$ is the spatial position vector, E is the neutron energy, Σ_t is the total neutron cross-section, Σ_s is the scattering neutron cross-section, Σ_f is the fission neutron cross-section, χ is the energy spectrum for neutrons born from fission, ν is the average number of neutrons born per fission, and k_{eff} is the eigenvalue representing the effective neutron multiplication factor. If the right hand side of Equation 1 is compressed into new total neutron source term $Q(\vec{r},\vec{\Omega},E)$, the form given in Equation 2 is reached.

$$\overbrace{\vec{\Omega}\cdot\nabla\psi(\vec{r},\vec{\Omega},E)}^{\text{streaming term}}+\overbrace{\Sigma_t(\vec{r},E)\psi(\vec{r},\vec{\Omega},E)}^{\text{absorption term}} = \overbrace{Q(\vec{r},\vec{\Omega},E)}^{\text{total neutron source term}} \quad (2)$$

An important aspect of these equations is the concept of a neutron cross-section. This value represents the probability of interaction between a neutron travelling at a certain speed (i.e., neutron energy E) and a target nucleus (i.e., the material through which the neutron is travelling). A neutron cross-section cannot be calculated by physics alone – rather, a combination of empirical data and quantum mechanical modelling must be employed in order to generate libraries of pointwise data for each target isotope of interest.

There exist a number of methodologies to solve the neutron transport equation for a particular reactor geometry and material composition. Each methodology involves a different set of assumptions that can be applied to Equation 1 in order to simplify it into a form that can be solved numerically. The Method of Characteristics (MOC) approach solves the equation along characteristic lines, thus discretizing the angular dependency of the equation into a set of linear tracks. In this manner, Equation 2 can be rewritten for a specific segment length s at a specific angle $\vec{\Omega}$ through a constant cross-section region of the reactor geometry as:

$$\frac{d}{ds}\psi(s,\vec{\Omega},E) + \Sigma_t(s,E)\psi(s,\vec{\Omega},E) = Q(s,\vec{\Omega},E) \tag{3}$$

An analytical solution to this characteristic equation can be achieved with the use of an integrating factor:

$$\psi(s,\vec{\Omega},E) = \psi(\vec{r_0},\vec{\Omega},E)e^{-\int_0^s ds'\Sigma_t(s',E)} + \int_0^s ds''Q(s'',\vec{\Omega},E)e^{-\int_{s''}^s ds'\Sigma_t(s',E)} \tag{4}$$

where r_0 is a reference location.

Similar to many other solution approaches to the Boltzmann neutron transport equation, the MOC approach also uses a "multi-group" approximation in order to discretize the continuous energy spectrum of neutrons travelling through the reactor into fixed set of energy groups G, where each group $g \in G$ has its own specific cross section parameters. With these key assumptions, the multi-group MOC form of the neutron transport equation can be written as:

$$\psi_g(s,\vec{\Omega}) = \psi_g(\vec{r_0},\vec{\Omega})e^{-\int_0^s ds'\Sigma_{tg}(s')} + \int_0^s ds''Q_g(s'',\vec{\Omega})e^{-\int_{s''}^s ds'\Sigma_{tg}(s')} \tag{5}$$

Note that there are a number of additional assumptions and minutiae not fully expressed in Equation 5, but are fully documented by Gunow et al.[5] and Boyd et al.[6].

1.2. 3D MOC Basics

Two-dimensional MOC calculations have long been used in reactor design and engineering as a medium-fidelity simulation method. While 2D assembly and core transport calculations have "reached industrial maturity" [7], they suffer from a loss of accuracy when compared with 3D models. For high fidelity modeling, the third dimension is necessary to correctly predict neutron leakage as well as axial power distributions in heterogeneous reactors. While MOC is easily extensible to three dimensions conceptually, the implementation of such an algorithm is difficult in practice as the computational requirements can be overwhelming [1]. To this end, an alternative approach has been developed to extend MOC into 3D.

The MOC solution to the neutron transport equation involves discretizing the equation along a set of characteristic *tracks* crossing the geometry of the reactor. Each specific track represents an angular neutron flux. At the outset of the calculation, the tracks are laid down over the reactor geometry and then subdivided into a series of *segments*, as shown in Figure 1, which correspond to the individual geometric regions (i.e., fuel, clad, water, etc.) along the characteristic track. This 2D track and segment information is generated for many different angles and starting points through the reactor at the outset of the computation. Extension to three dimensions is in principle possible by extending the same process to include all the tracking and segment information for the full 3D geometry, but the memory cost would be prohibitive. Our formulation of the 3D MOC algorithm instead uses an extruded geometry in the axial direction, which saves memory by storing only the explicit 2D track and segment information, and then performing on-the-fly ray tracing along stacked z-ray tracks (at many different angles) that extend the computation into the 3rd dimension, as depicted in Figure 2. This method significantly reduces the track information memory requirements for 3D MOC at only a modest computational cost, thanks in large part to the nearly homogeneous nature of reactor geometry in the axial direction.

1.3. Method of Characteristics with Quadratic Source Regions

The core idea behind the MOC algorithm is that along a characteristic track, the complicated multi-group neutron transport equation can be reduced

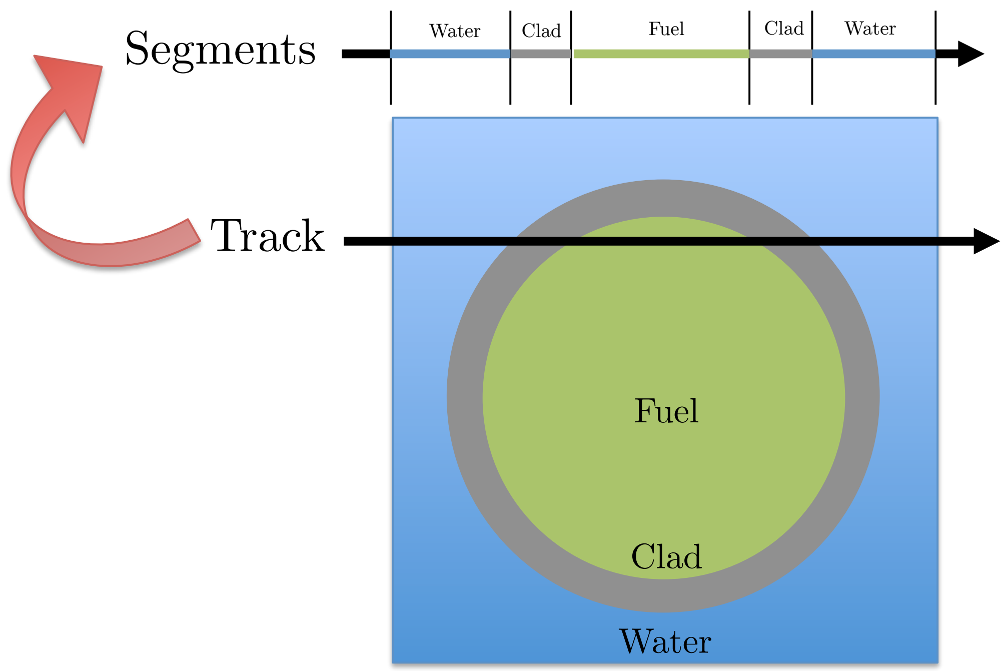


Figure 1: A single MOC track through a simplified reactor pin cell 2D geometry. This example shows the relationship between a constructive solid geometry, a track, and the segments which compose the track. An "intersection" is a further subdivision of a segment representing a single energy group of the flux.

to an ordinary differential equation that can be solved analytically [6]. This implies that the angular neutron flux ψ can be easily computed along chosen directions. The scalar flux ϕ of each region, which is necessary to compute reaction rates, can be computed by adding the contributions of each angular flux track that crosses through the region. Mathematically this can be expressed as

$$\phi = \frac{4\pi}{V} \sum_{k \in V} w_k l_k \bar{\psi}_k \tag{6}$$

where the index k refers to the track number, w_k refers to the weight of track k, l_k refers to the segment length of track k that passes through the region, $\bar{\psi}_k$ refers to the average angular flux of track k through the region, and V refers to the volume of the scalar flux region. The weights w_k are dependent on the specific azimuthal quadrature selection, and correspond the angular space "owned" by the track. These weights can be slightly different between

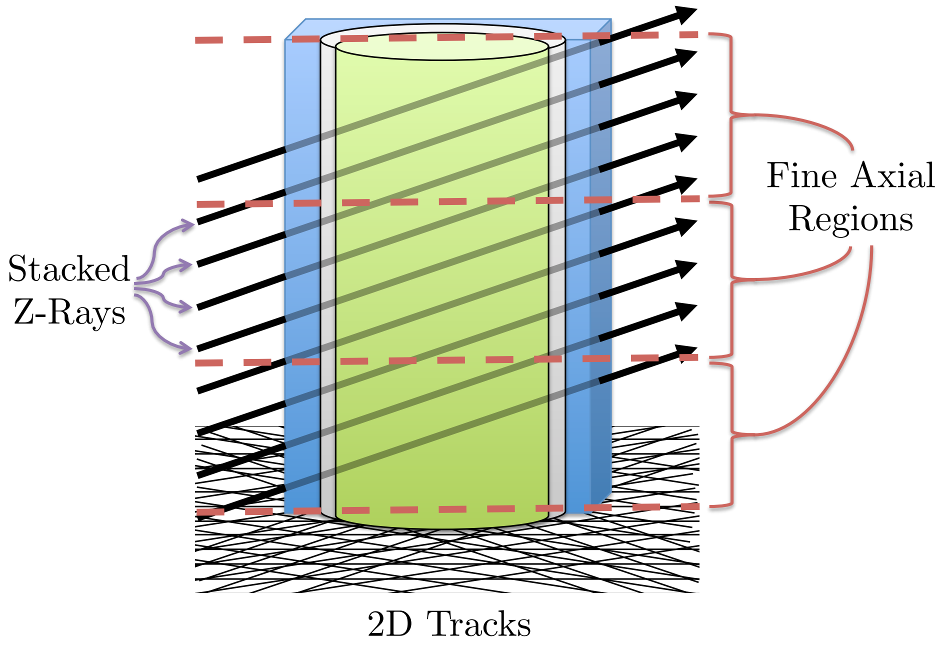


Figure 2: Cutaway of a simplified reactor pin cell 3D geometry. This example shows how our algorithm uses an explicit 2D tracking file and performs on-the-fly ray tracing in the axial direction.

tracks due to cyclical tracking requirements affecting the particulars of the track laydown routine. The particulars of the weighting scheme are described in depth by Shaner et al.[8] as well as Boyd et al.[6].

Therefore, the core function of an MOC simulation code is the numerical sweeping of discrete angular flux tracks across the reactor geometry. In this process, we must perform several computations for each geometric segment within the track, namely the attenuation of the angular flux ψ for a particular energy group along a track and the contribution of that angular flux to the scalar flux ϕ. After these calculations are completed, a new neutron source is calculated from the scalar flux tallies. As described by Gunow et al.[5], the neutron source is approximated as quadratic in the axial direction for each constant cross-section region. Taking advantage of the fact that large coarse axial regions can be approximated as having constant cross-sections, each coarse axial region is split into several fine axial regions. The source contribution in each fine axial region can be directly formed on-the-fly during the transport sweeps with the source magnitude calculated by fitting a quadratic function to the sources neighboring fine axial regions. To simplify the process and lessen computational requirements, the fitting routine only involves the fine axial region being traversed and the neighboring fine axial regions, as these three points can be used to define a quadratic function that can be directly computed without any fitting routine.

With the implementation of higher order axial sources, the total number of intersections necessary to converge the solution are greatly reduced. However, the number of floating point operations (FLOPS) per intersection greatly increases. Overall, usage of the quadratic source approximation is expected to result in a significant net decrease in program runtime [5]. To analyze the computational performance, we first need to consider the MOC equations over the constant cross-section fine axial regions across which the equations are applied. We are primarily concerned with two computations: the attenuation of the angular flux ψ for a particular energy group along a track and the contribution of that angular flux to the scalar flux ϕ of the fine axial region. To determine the angular flux attenuation, ψ is defined as a function of distance s traversed along the track in the fine axial region with total cross-section Σ_t. The normal approximation of a spatially constant source q_0 leads to simple exponential attenuation given by Equation 7.

$$\psi(s) = \psi(0)e^{-\Sigma_t s} + \frac{q_0}{\Sigma_t}\left(1 - e^{-\Sigma_t s}\right) \tag{7}$$

In contrast, if a quadratic approximation of the neutron source is introduced, defined by Equation 8

$$q(s) = q_0 + q_1 s + q_2 s^2 \tag{8}$$

where s is the distance traveled along a characteristic track, and q_1 and q_2 are the first and second order spatial coefficients of the scalar flux in the z-axis direction. The quadratic approximation makes the angular flux attenuation across each track much more computationally demanding, as seen in Equation 9.

$$\begin{aligned}\psi(s) = \psi(0)e^{-\Sigma_t s} + \frac{q_0}{\Sigma_t}\left(1 - e^{-\Sigma_t s}\right) + \frac{q_1 \cos\theta}{\Sigma_t^2}\left(\Sigma_t s + e^{-\Sigma_t s} - 1\right) \\ + \frac{q_2 \cos^2\theta}{\Sigma_t^3}\left(\Sigma_t s\left(\Sigma_t s - 2\right) + 2\left(1 - e^{-\Sigma_t s}\right)\right).\end{aligned} \tag{9}$$

In addition to determining the angular flux attenuation, notice that in Equation 6 the average angular flux across the region is needed to determine the scalar flux. For the flat source approximation, this can be simply computed with the relation given in Equation 10

$$\bar{\psi} = \frac{\psi(0) - \psi(s)}{s\Sigma_t} + \frac{q_0}{\Sigma_t} \tag{10}$$

This adds trivial computational work since the difference in angular flux $\psi(0) - \psi(s)$ can be easily derived from the computation of the angular flux attenuation and the constant q_0/Σ_t term can be added to each scalar flux tally after the transport sweep. For quadratic sources, the calculation of the average angular flux is not as simple, with the required calculation given in Equation 11.

$$\begin{aligned}\bar{\psi} = \frac{1}{s}\Bigg[\frac{1}{\Sigma_t^2}\left(q_0\Sigma_t s + \left(\Sigma_t\psi(0) - q_0\right)\left(1 - e^{-\Sigma_t s}\right)\right) \\ + \frac{q_1 \cos\theta}{2\Sigma_t^3}\left(\Sigma_t s\left(\Sigma_t s - 2\right) + 2\left(1 - e^{-\Sigma_t s}\right)\right) \\ + \frac{q_2 \cos^2\theta}{3\Sigma_t^4}\left(\Sigma_t s\left(\Sigma_t s\left(\Sigma_t s - 3\right) + 6\right) - 6\left(1 - e^{-\Sigma_t s}\right)\right)\Bigg]\end{aligned} \tag{11}$$

Notice that in both the computation of the angular flux attenuation and average scalar flux for quadratic sources, there are three exponential terms instead of just one, as is the case with the flat source approximation. With efficient computation, this fact alone does not add much computational work as the exponential only needs to be computed once and then reused. However, there is a significant increase in the number of multiplications, causing a large increase in the required number of FLOPS. Due to the number of terms in the equation, there is also increased register pressure as the number of values needed to be stored exceeds the space available in the CPU's registers. Therefore, great care must be taken in efficiently computing the attenuation and contribution to the scalar flux.

1.4. Target Simulation Problem

In this study, we will examine the parallel computational performance aspects of the 3D MOC algorithm on a variety of high performance computing platforms. In order to form an accurate assessment, it is important to pick a reasonable target problem for our tests that can provide a benchmark for real-world usage scenarios for a full application. While there are a wide variety of important reactor problems to solve, we are specifically targeting high-fidelity full core reactor simulation applications, complete with several hundred fuel nuclides and 100+ energy groups. A typical target simulation is well represented by the BEAVRS community benchmark [9], which is composed of parameters given in Table 1. As discussed by Gunow et al.[5], BEAVRS would require approximately 75 TB of total data using our formulation of the 3D MOC transport algorithm, which can be domain decomposed onto a current generation supercomputer such as the IBM Blue Gene/Q *Mira* using approximately 5,780 nodes [5].

Table 1: Anticipated parameters for a converged 3D MOC solution of a PWR core.

Parameter	Dimension
Geometry layout	17×17 assemblies
Assembly width (x and y)	21.24 cm
Assembly height (z)	400 cm
Coarse axial heterogeneity spacing	10 cm
Fine axial regions per coarse axial region	5
Number of energy groups	128
Radial ray spacing	0.05 cm
Axial ray spacing	0.25 cm
Number of azimuthal angles	64
Number of polar angles	10

2. Application

To investigate on-node and mutli-node performance and scaling characteristics of the 3D MOC algorithm, we have developed two different proxy-applications – SimpleMOC and SimpleMOC-kernel – designed to mimic the key performance characteristics of a full 3D MOC production application. They retain the essential performance-related computational conditions and tasks of fully featured reactor core 3D MOC neutron transport codes, yet at a fraction of the programming complexity of a full application. These proxy-applications provide a basis for evaluating the algorithm on various architectures, and serve as a first step towards building a full featured production 3D MOC solver. In this section, we describe the purpose of each application and provide an outline of the algorithms, data structures, and access patterns that they implement.

2.1. SimpleMOC

The SimpleMOC proxy application mimics the runtime of a full scale 3D MOC reactor simulation when domain decomposed across many compute nodes, as is commonly done in supercomputer calculations. This allows for the computational sweeping kernel to be studied, as well as its relative costs compared to the inter-node communication kernel that must be executed between each sweep. This provides a much simpler and far more transparent platform for testing the algorithm on different architectures, making alterations to the code, and collecting hardware runtime performance data.

SimpleMOC is implemented in C, with inter-node parallelism expressed in MPI and on-node shared memory threading expressed in OpenMP. It is an open source code and is available online [10].

2.1.1. Algorithm

Algorithm 1 outlines the SimpleMOC implementation of the 3D MOC algorithm as a top-down pool of independent tracks that can be dynamically assigned to processors to ensure optimal load balancing. This key feature is expressed as a parallel OpenMP for loop utilizing dynamic scheduling. A second key feature of this algorithm is the wide SIMD vector forming the inner loop over energy groups, which maps extremely efficiently onto modern computing architectures.

Each track is composed of a series of segments that must be processed sequentially in-order by the processor (as shown in Figures 1 and 2). The inner loop of the algorithm involves three computations: an exponential evaluation, the attenuation of the angular flux ψ for a particular energy group along a track, and the contribution of that angular flux to the scalar flux ϕ of the fine axial region. The added computational complexity inherent in the quadratic source approximation allows for a significant increase in fidelity, and moreover can easily be vectorized by the compiler. Note that Algorithm 1 is slightly simplified – the actual implementation includes several additional loops where Line 1 is found, representing polar angles and stacked z-rays, though collectively these variables are independent and could also be expressed as simply a pool of tracks as shown in Algorithm 1.

Algorithm 1 SimpleMOC 3D MOC Algorithm Formulation (Single Sweep)

```
 1: for all Tracks do                                    ▷ thread level parallelism
 2:     Load Boundary Flux Data
 3:     for all Segments do
 4:         for all Energy Groups do               ▷ SIMD vector level parallelism
 5:             Exponential Evaluation
 6:             Attenuate Flux
 7:             Contribute Scalar Flux to Fine Axial Region              ▷ atomic
 8:         end for
 9:     end for
10: end for
11: Exchange Boundary Flux Data with Neighbors
```

While tracks may in general be processed independently, the contribution of scalar flux to source regions must be an atomic or mutex locked operation, as multiple tracks (i.e., threads) may be passing through a given source region at any point in time. In SimpleMOC, this has been implemented using an array of locks corresponding to each individual source region. Whenever a thread is ready to write-back data to a source region, that region's mutex lock is set until all energy group fluxes have been updated. This operation comes with minimal overhead as there are far more source regions than there are threads, making lock contention trivial. In practice we observed only about 5% overhead from locking.

Finally, at the end of each computational sweep through the full geometric domain, a number of small bookkeeping functions are executed and boundary flux data is exchanged between nearest neighbor nodes using MPI. Prior work by Gunow et al. has shown that communication costs, as measured by weak scaling on the IBM Blue Gene/Q supercomputer *Mira*, are extremely low and account for only a very small percentage (less than 5%) of the program's overall runtime [5].

2.1.2. Data Structures & Access Patterns

There are two key data structures that play a critical role in the performance of SimpleMOC. The first is the source region flux array. When the problem is decomposed as in Table 2, we anticipate approximately 36 MB of data to be required to store all the relevant fluxes. Each time a segment is processed, the geometrical source region that the segment resides in must be accessed and the accompanying source flux vector (i.e., a `float` value for each energy group) must be loaded. As the source regions of two neighboring segments are unlikely to be co-located in memory (see section 2.2.1), the resulting access pattern of the source arrays have low spatial and temporal locality. Thus, it is important to retain the source region data array in cache, in order to reduce latency costs associated with random reads from main memory. The memory footprint of the source region array maps well onto most, but not all, HPC node architectures as they often feature 20–40 MB of cache[1].

Table 2: SimpleMOC example decomposition for the BEAVRS benchmark

Parameter	**Value**
Nodes	5,780
Source Region Flux Array Size per Node	36 MB
Border Flux Array Size per Node	12 GB

The second key data structure is the boundary flux array. This structure is very large, roughly 12 GB in our example discretization (Table 2), and holds all the incoming and outgoing track flux that must be exchanged via MPI between neighbor nodes after each computational sweep. While this data structure is extremely large, it is rarely accessed. Each thread processes a single track at once, first loading in the incoming flux vector for the track from the boundary flux array (i.e., a `float` value for each energy group), and then processing the flux across all segments in the track (about 120 of them) before storing the resulting flux to the outgoing section of the boundary flux array. This infrequent and predictable access pattern means that the border flux arrays are not important to keep in cache and can instead use DRAM memory. Future exascale systems are likely to feature slow but large NVRAM modules (providing nodes with up to several TB of high latency memory), which should provide another good alternative given the boundary flux array's insensitivity to high memory latencies.

[1]The exception is the Intel Phi architecture, such as the 7120a, which only features 512 KB of effective last level cache.

2.2. SimpleMOC-kernel

Table 3 shows the call stack timing breakdown for SimpleMOC when run on a modern CPU architecture (a dual socket, 36-core Intel Xeon E5-2699v3 Haswell node). The majority of the runtime in SimpleMOC is spent performing the computational sweep (i.e., attenuating neutron fluxes across segments). With this in mind, combined with the knowledge that inter-node communication costs are minimal for our target problem [5], we have extracted a kernel application "SimpleMOC-kernel" which removes the MPI communication functions and simplifies the looping structure of the application into a simpler form. This was done to better isolate the computationally important aspects of the algorithm in order to allow for easier profiling and easier porting to new HPC node architectures.

Table 3: Call Stack Time Breakdown by Function in SimpleMOC

% Time	**Function**	**Lines in Algorithm 1**
82.11%	Attenuate Fluxes	4, 6, 7
11.79%	Exponential Evaluation	5
3.84%	Tracking Loops Above Attenuation	1, 2, 3
2.26%	Other	11

SimpleMOC-kernel is implemented in C with shared memory threading expressed in OpenMP. It is also implemented in CUDA in order to be run on NVIDIA GPUs. Overall, the differences between the CPU and GPU algorithms are fairly trivial, as the CPU algorithm is already well expressed in thread level (independent) and vector level (SIMD) groupings, which maps extremely easily onto GPU architectures. Conversion to CUDA was done by grouping segments together (100 at a time) and assigning them to a CUDA block. Each thread within the block then handles a single energy group.

Both CPU and GPU code versions have been aggressively optimized for their respective architectures, though not beyond what would be reasonably expected of performance-oriented scientific programmers. SimpleMOC-kernel was also ported to OCCA and OpenACC [11], though only the CUDA version is presented in our GPU analysis. SimpleMOC-kernel is an open source code and is available online [12].

2.2.1. Algorithm & Data Structures

Algorithm 2 depicts a simplified version of the 3D MOC algorithm as implemented in SimpleMOC-kernel [12]. As the attenuation of neutron fluxes accounts for over 90% of the runtime of the application (the exponential evaluation is part of this), the algorithm can be simplified to remove tracks into a simpler series of segments. While a production MOC solver is required to attenuate a flux across a specific set of segments sequentially (composing a track, as in Figure 1), the actual computational flavor is similarly expressed as a loop over randomized independent segments. While this is in a sense unphysical, the only difference from a computational performance standpoint is that the importing and storage of boundary flux at the beginning and ending of each track is not captured by the independent segment assumption. However, this difference can reasonably be ignored, as there are approximately 120 segments per track in our target problem discretization (i.e., the BEAVRS benchmark), requiring about 150,000 FLOPS per border flux cache line read. Instead, the state flux between segments can more simply be preserved on the local thread rather than by track. This simplification from tracks to independent segments significantly reduces the programming bookkeeping and greatly reduces the length and complexity of the source code.

Algorithm 2 SimpleMOC-kernel Simplified 3D MOC Abstraction

```
1: for all Segments do                                  ▷ thread level parallelism
2:     Randomly Sample Quadratic Source Region Index
3:     Randomly Sample Fine Axial Interval Index
4:     for all Energy Groups do                     ▷ SIMD vector level parallelism
5:         Exponential Evaluation
6:         Attenuate Flux
7:         Contribute Scalar Flux to Fine Axial Region                    ▷ atomic
8:     end for
9: end for
```

A full production MOC solver also associates a specific source region to each segment. Even though a full MOC solver processes segments within a track in-order, the source regions of two neighboring segments are unlikely to be co-located in memory due to the complexities of mapping a constructive solid geometry set of source regions into serial memory. Even if optimizations were performed to the data layout, it would likely be impossible to achieve significant source locality along segments in a track due to many segments at unpredictable angles and z-heights passing through the same source region. In effect, accessing of source regions during a computational sweep appears to the computer as a "random walk" through memory. Therefore, a similar level of random access of the source regions is preserved by randomizing the several key definition parameters for each segment (i.e., quadratic source region and fine axial interval). Rather than accessing a stored source index for each segment as in a full MOC solver, SimpleMOC-kernel instead selects a random quadratic source index ID and fine axial interval ID for each segment. This simplification greatly reduces the amount of bookkeeping in the source code implementation while still retaining the same computational characteristics as a full 3D MOC solver.

2.2.2. Kernel Verification

The simplified kernel version of this algorithm captures the full computational flavor of the full application very well, matching the performance of the full application almost exactly. Figure 3 shows the strong scaling of both applications on a dual socket, 36-core (72 thread) Intel Xeon E5-2699v3 Haswell node. The speedup when running all 72 threads on the node is only 4.1% different between SimpleMOC and SimpleMOC-kernel, which is remarkably close given that the central control-flow portions of SimpleMOC-kernel is about 8 times smaller in terms of lines of code compared to the full SimpleMOC application. Additionally, we compared performance between the applications directly via a common figure of merit – the amount of time it takes to attenuate a single energy group of flux across a single geometric segment (i.e., the time per *intersection*). This figure of merit was observed to be extremely close between the two applications – SimpleMOC ran at 0.33 ns while SimpleMOC-kernel ran at 0.28 ns at 36 threads. Overall, this shows that SimpleMOC-kernel manages to very accurately represent the key computational parameters of the full application, thus making it far easier to understand from a performance perspective and to port to new architectures or coding languages.

3. Performance & Bottleneck Analysis

In order to evaluate the speed and future on-node scalability of the 3D MOC algorithm on various architectures, a series of tests was performed using performance data as well as hardware performance counters.

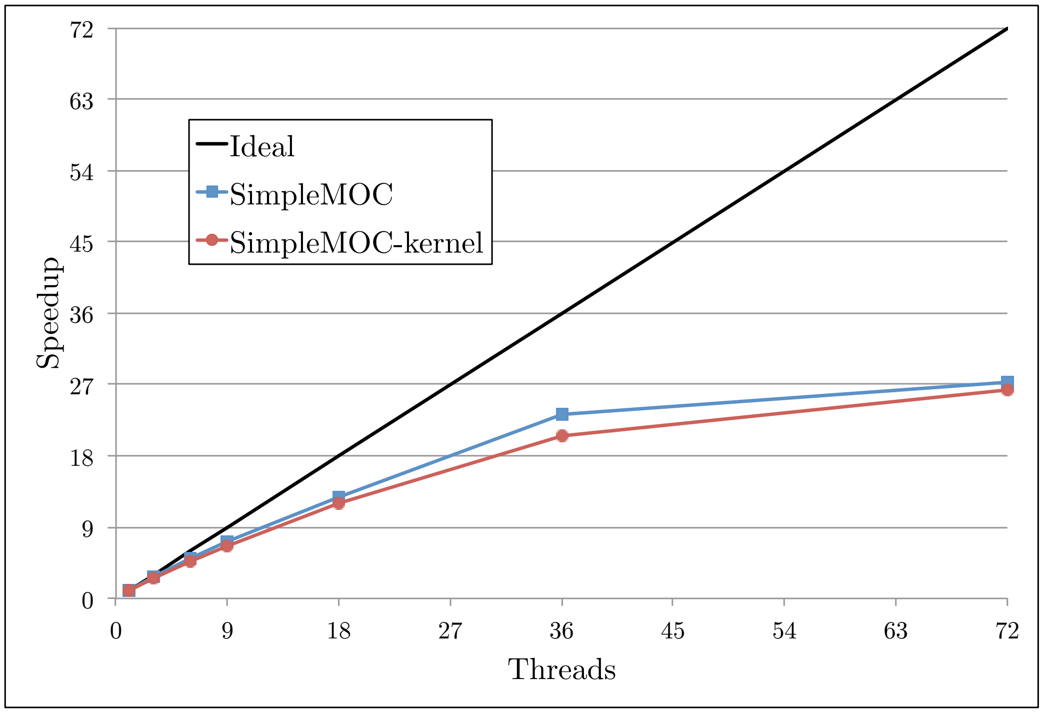


Figure 3: SimpleMOC vs SimpleMOC-kernel speedup on 36-core (72 thread) Intel E5-2699v3 Haswell node.

3.1. Performance Comparison

A comparative performance study using SimpleMOC-kernel was performed on six different platforms. In each test, two billion segments were processed for 128 energy groups and the resulting figure of merit (time per intersection) was reported. The results are shown in Figure 4. As can be seen, the NVIDIA GPUs all perform extremely well, with the TitanX outperforming the other architectures at 0.081 ns. The dual socket Intel E5-2699v3 Haswell node also performs well, at 0.218 ns. The Intel Phi 7120a node performs worse, only achieving a time per intersection of 0.383 ns. Finally, the dual socket Intel E5-2650 Sandy Bridge node performs the worst, at 0.627 ns, but represents a more common platform deployed in many current CPU based clusters.

Note that the TitanX platform is only capable of such high performance if the bulk of the computation is performed using single precision floating point values. As it is a consumer grade GPU, it lacks the ability to perform double precision calculations efficiently, unlike all other platforms in this study including the NVIDIA K40m and K20c. However, for our target application (MOC style reactor calculations), a significant portion of the computational work (such the inner loop, wherein angular flux is attenuated across a segment, as in Equations 9 and 11) will not be sensitive to single precision errors as no large accumulations take place and floating point arithmetic is likely to stay within a few orders of magnitude. Other less common operations, such as atomic writes into the source region scalar fluxes, may be more sensitive to floating point error. For this purpose, SimpleMOC-kernel uses double precision for the scalar flux arrays. As these double precision operations only account for a small percentage of the total floating point computations in the code however, the code still runs very efficiently even on the TitanX architecture that has limited double precision resources.

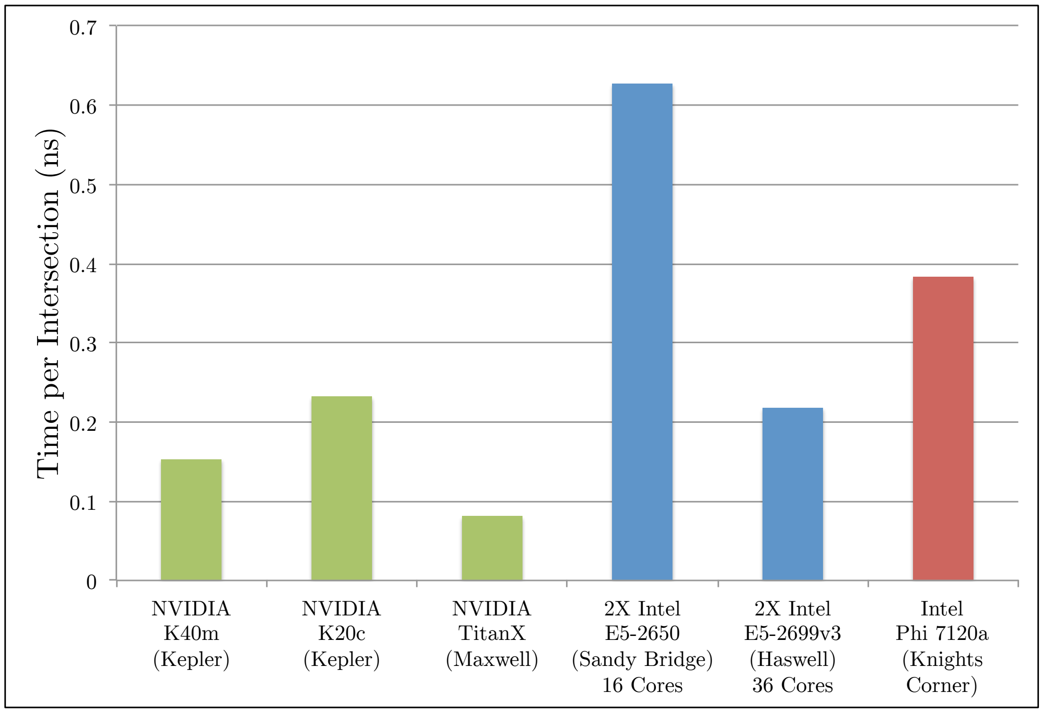


Figure 4: SimpleMOC-kernel performance comparison on various architectures. Performance is measured as time (ns) per intersection (lower is better). Performance timings were taken using all computational resources (i.e., all cores, full hyperthreads) from a node.

Additionally, floating point operations per second (FLOPS) were also measured using PAPI [13] performance counters on CPU and NVIDIA's NVProf performance metrics on GPU. This data, given in Figure 5, shows that the TitanX GPU is able to achieve an extremely high FLOP rate of 1,335 GFLOPS, which for comparison is over 8.7 times higher than the 2X Intel E5-2650 Sandy Bridge node at 153 GFLOPS. However, it is important to note that the theoretical FLOP maximums on these systems are very different due to the large differences in hardware architectures such as different die sizes, power consumption, hardware threading, and vector widths. Even though the kernel is running extremely efficiently on the Sandy Bridge CPU, achieving over 63% of peak achievable FLOPS (as measured by the LINPACK benchmark [14]), it is still getting outperformed overall by the GPUs due to the greatly improved FLOP capabilities of those architectures.

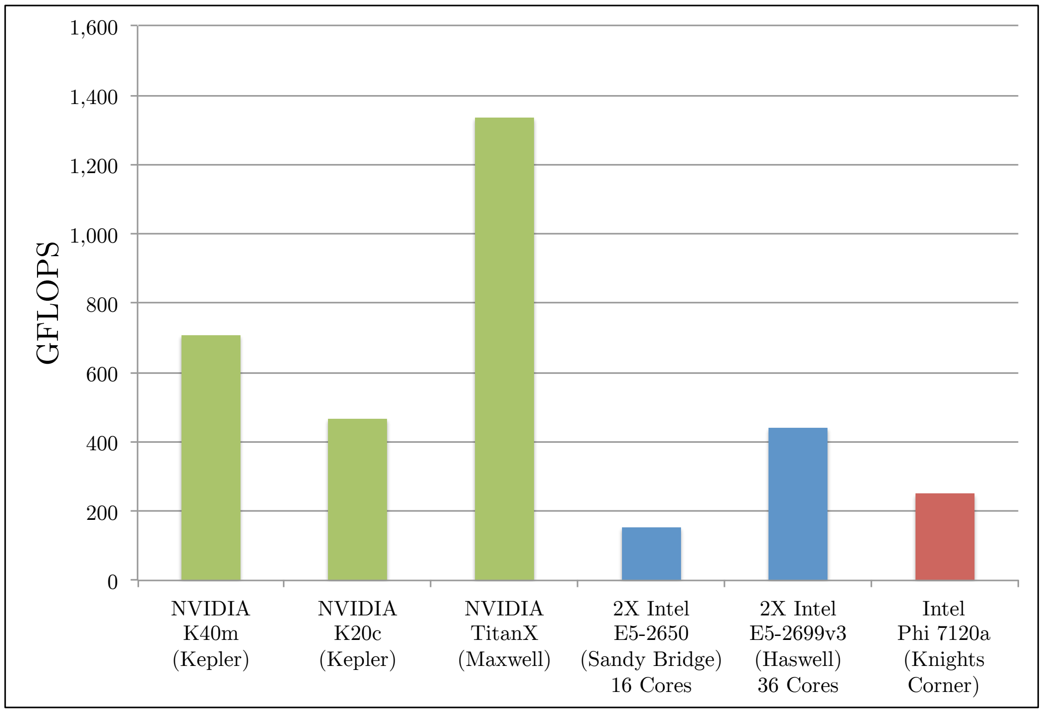


Figure 5: SimpleMOC-kernel – Floating point operations per second on various architectures.

Achieving high floating point throughput on any modern architecture requires the usage of the vector SIMD floating points units that are present on the CPU, Phi, and GPU. The 3D MOC algorithm, as depicted in Algorithm 2, takes advantage of this architectural characteristic by composing the inner loop across energy groups rather than by polar or azimuthal angle. This is highly advantageous as a full core computation will have at least a hundred energy groups, allowing for a high vectorization efficiency. A study of the impact of the number of energy groups (i.e., the SIMD vector width) was performed on CPU, GPU, and Phi architectures, as seen in Figure 6. Increasing the number of energy groups results in efficiency gains of one to two orders of magnitude. The NVIDIA TitanX GPU is able to achieve reasonable vector efficiency with the fewest energy groups – reaching within 30% of its fastest time per intersection at only 32 energy groups, compared to 56 energy groups for the 2X E5-2699v3 Haswell CPU node and 144 for the Intel 7120a Phi. Note that the slight degradation of performance of the Intel E5-2699v3 Haswell node is a result of the increase in the size of the source region flux array that accompanies the increase in the number of energy groups. As discussed in Table 2, these source arrays are approximately 36 MB for 128 energy groups, but double to 72 MB when 256 groups are used. The Haswell's last level cache is 45 MB, making the transition towards 256 energy groups progressively increase the cache miss frequency.

The extreme impact of the number of energy groups on the efficiency of the calculation is an important result. Experiments using only several energy groups will have a very different performance profile relative to full scale simulations. In this paper, all performance data was collected using 128 energy groups.

3.2. Bottlenecks

While Figure 4 clearly shows the performance disparity between architectures, it is critical to understand the source of these differences. Some issues are likely to be remedied by future micro-architectural development while others could potentially get worse with time. Famously, the "von Neumann" bottleneck [4] is especially dangerous for architecture lines where FLOP capacities follow Moore's Law while memory bandwidth improves only marginally. This von Neumann bottleneck causes the FLOP units to rapidly become starved. Thus, it is important to identify the reasons why we are seeing the performance trends in Figure 4 and to evaluate the potential effects of these issues on future architectures.

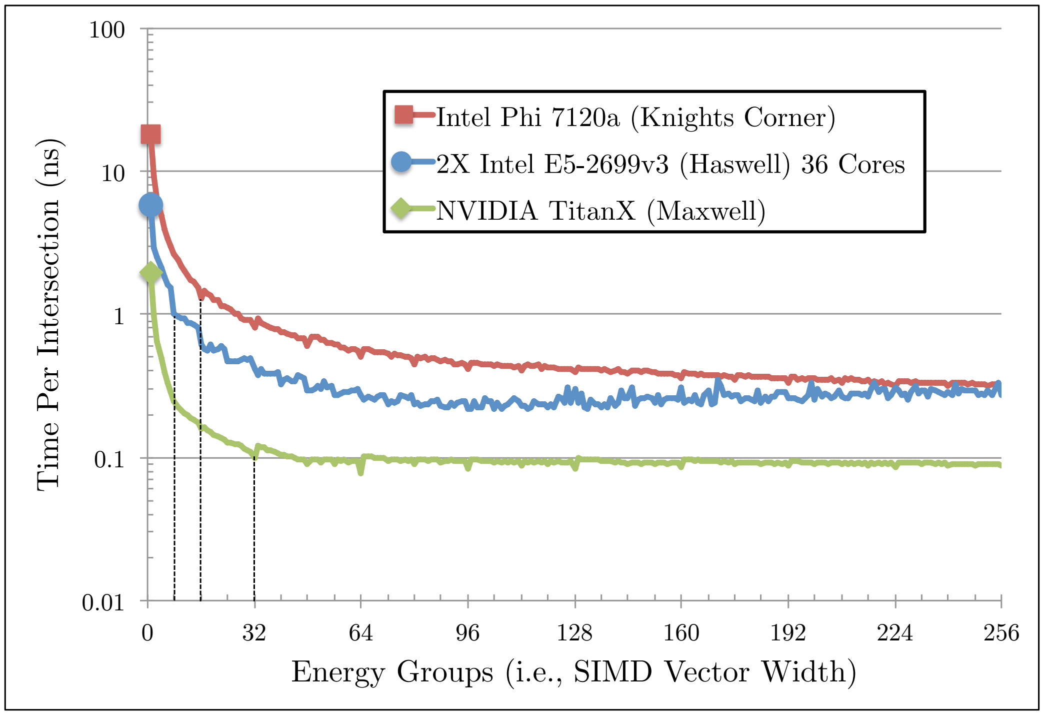
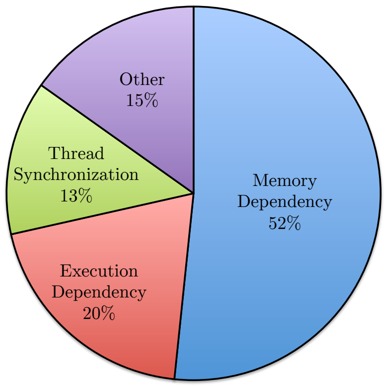


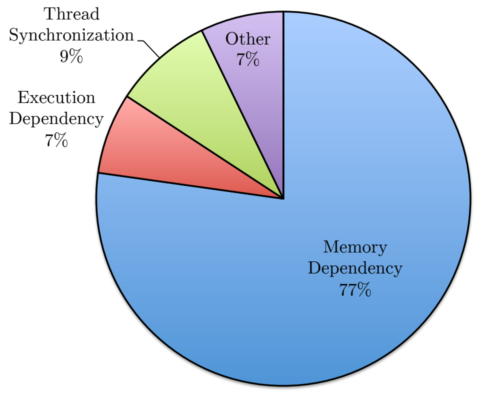
Figure 6: SimpleMOC-kernel – Performance (time per intersection) versus the number of energy groups considered in the calculation. Computation across energy groups forms the inner loop of the kernel and therefore determines the efficiency of vectorization. Close inspection of the plot reveals a minor periodicity corresponding to the vector widths of each machine (32 wide for the GPU, 16 wide for the Phi, and 8 wide for the CPU), as indicated by the dotted vertical lines.

3.2.1. CPU & GPU

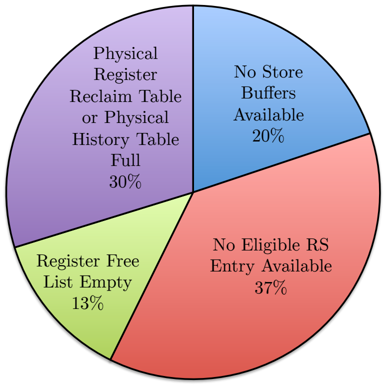
We used PAPI [13] performance counters on the CPU and NVIDIA's NVProf performance metrics on the GPU to determine the cause of stalled cycles incurred during the runtime of SimpleMOC-kernel. Figure 7 compares a 16-core Intel E5-2650 Sandy Bridge node to an NVIDIA TitanX GPU. The performance bottleneck for the GPU is clearly identified as a memory latency or bandwidth bottleneck, as stalled cycles are overwhelmingly due to memory dependencies. Conversely, the CPU stall graph shows that a wider variety of stall factors are at play. However, most of these stalls indicate register pressure. Together with the high values seen in Figure 5, this indicates that the source of the bottleneck is operands that are delayed when finding their way between the level 1 (L1) cache and the CPU registers due to the register scheduling mechanisms being overwhelmed. Any further optimizations would likely involve manual management of the registers by directly writing assembly code, though potential for improvements is small considering the already extremely high percentage of peak performance achieved on the CPU.



(a) GPU (K20c)



(b) GPU (TitanX)



(c) CPU (Sandy Bridge)

Figure 7: Stalled Cycle Analysis - This figure shows the various reasons for stalled cycles on CPU and GPU. As shown, the GPUs are bottlenecked by memory dependency issues (i.e., latency or bandwidth), while the CPU is stalled by a wider variety of factors. The CPU stall breakdown indicates that register pressure is the likely bottleneck.

3.2.2. Intel Phi

SimpleMOC-kernel performance on the Intel Xeon Phi appears to be limited by the last-level cache (LLC). On the 7120a model, this is a level 2 (L2) cache. The Xeon Phi has 61 cores, each of which has a local 512 KB L2 cache. These are connected in a bidirectional ring to yield the combined 30.5 MB of shared L2. While this appears to be sufficient memory to hold most of the source region data in cache, in reality each core's local L2 can only store a small portion of the data. This means when a thread does not find its required data in its local L2, it must fetch from another core's L2 through the ring interconnect. While this is technically not a cache miss, it is still a significant penalty and has a similar impact as a full cache miss to DRAM.

One way to measure this is the "estimated latency impact," which is an approximate measure of the number of cycles spent on a (level 1) L1 cache miss. For SimpleMOC-kernel's transport sweep, we measured this to be 109.8. Normal L2 access latency is 21 cycles, and main memory latency is about 300 cycles. Furthermore, L1 cache miss rate was measured to be 17%, which means a significant amount of time spent fetching from non-local L2 and main memory. Due to the effectively random access patterns of source region data, data locality for each core is difficult to achieve, magnifying the penalty of L1 misses.

Vectorization was evaluated using the metric "vectorization intensity," which is a measure of the vector processing unit (VPU) efficiency. For ideal vectorization, this quantity is 8 for double precision, and 16 for single precision. We obtained a value of 14.8 for the transport sweep, which shows the effective use of the VPU (as is also demonstrated in Figure 6).

3.3. Power

While Figure 4 clearly shows that the GPUs achieve superior performance compared to CPU or Phi, the question remains as to whether or not they are achieving this performance in a power efficient manner. As current supercomputers such as the Tianhe-2 already use 17.8 MW of electricity, and many exascale designs are limited to power budgets around this level, it is critical to normalize any performance numbers by power. In order to do this, we measured the power of two GPUs (the K20c and the TitanX), the Intel Phi 7120a, and the dual socket Intel E5-2650 Sandy Bridge node[2]. Power

[2]Note that power metrics were not collected for the Haswell or K40m nodes. This is due to difficulties associated with PAPI RAPL support on the Haswell architecture and

was measured on CPU via Intel running average power limit (RAPL) power counters via PAPI, which allowed us to measure the total power used by the CPU+DRAM of the node over the course of the computation. GPU power was measured using the NVIDIA "nvidia-smi" utility, which provided similar power data over the course of the computation for the entire GPU card. Intel Phi power was measured using the Intel "micsmc" utility. For all architectures, power readings were taken every 100 ms while computing the flux attenuation over 1 billion segments.

As we can see in Figure 8, the TitanX is by far the most power hungry of the architectures tested, reaching around 237 Watts compared to the 145 Watts averaged by the Intel Sandy Bridge node. However, given that the amount of time taken to process the 1 billion segments was far less for the TitanX and K20 GPUs, the total amount of energy spent during the computation is the critical metric of importance. The power data from Figure 8 was integrated to determine energy per billion intersections, as shown in Figure 9. Here we can see that the TitanX GPU performs the best, achieving a 4.8x power efficiency advantage over the Intel Sandy Bridge CPU platform, and a 4x advantage over the Intel Phi node.

3.4. Hardware Dollars

While power efficiency is the most critical metric to consider in high performance computing and supercomputer design, it is also important to consider the direct hardware cost of a compute node. Even though the NVIDIA TitanX is the highest performing and most power efficient card, it is also surprisingly the cheapest node design ($1,100) of the six that we compared in this study (as measured by retail prices on Amazon.com in June of 2015). When performance is normalized by hardware price (in terms of $ per billion intersections/sec), as shown in Figure 10, we find that the NVIDIA TitanX is able to best Intel's highest performing offering (the dual socket E5-2699v3 Haswell node) by over a factor of 23x.

The extreme price efficiency of the TitanX GPU is caused by the fact that the TitanX is a consumer grade device. Notably, it lacks the ability to perform double precision calculations efficiently unlike all other platforms tested in this analysis. However, by making sparing use of doubles for key variables, performance and accuracy can still be maintained on the TitanX

limited access to the K40m node.

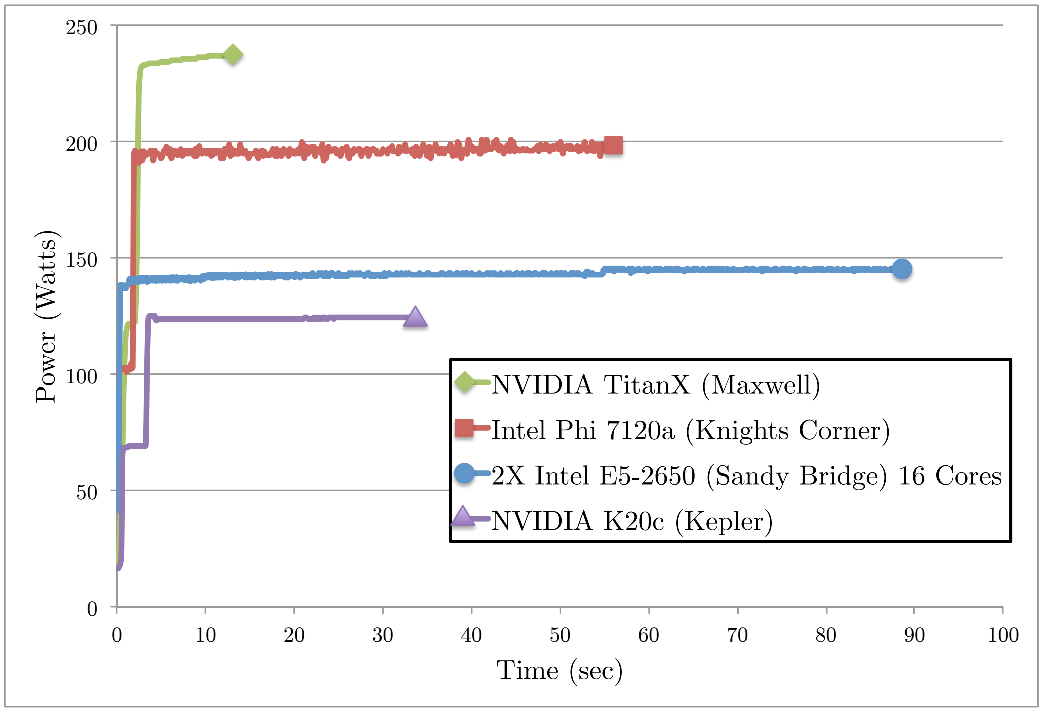


Figure 8: SimpleMOC-kernel power (Watts) performance comparison on various architectures. Power is measured for 1 billion segments (including program initialization) on all three architectures.

architecture as discussed in subsection 3.1.

4. Conclusions

In this study, we have developed a performance abstraction for the 3D Method of Characteristics (MOC) algorithm in the context of high fidelity full core nuclear reactor simulations. To this end, we developed two different proxy-applications, SimpleMOC and SimpleMOC-kernel, that capture the multi-node and single node performance and scaling characteristics of the 3D MOC algorithm. SimpleMOC-kernel was also ported to the GPU platform, and a wide variety of architectures (including CPU, GPU, and the Intel Phi) were compared. Our mini-apps were able to show extremely high performance due to 1) usage of a task-based pool of independent MOC tracks allowing for efficient dynamic load balancing and 2) the placement of energy groups at the inner loop of the computation allowing for the wide SIMD vector units of

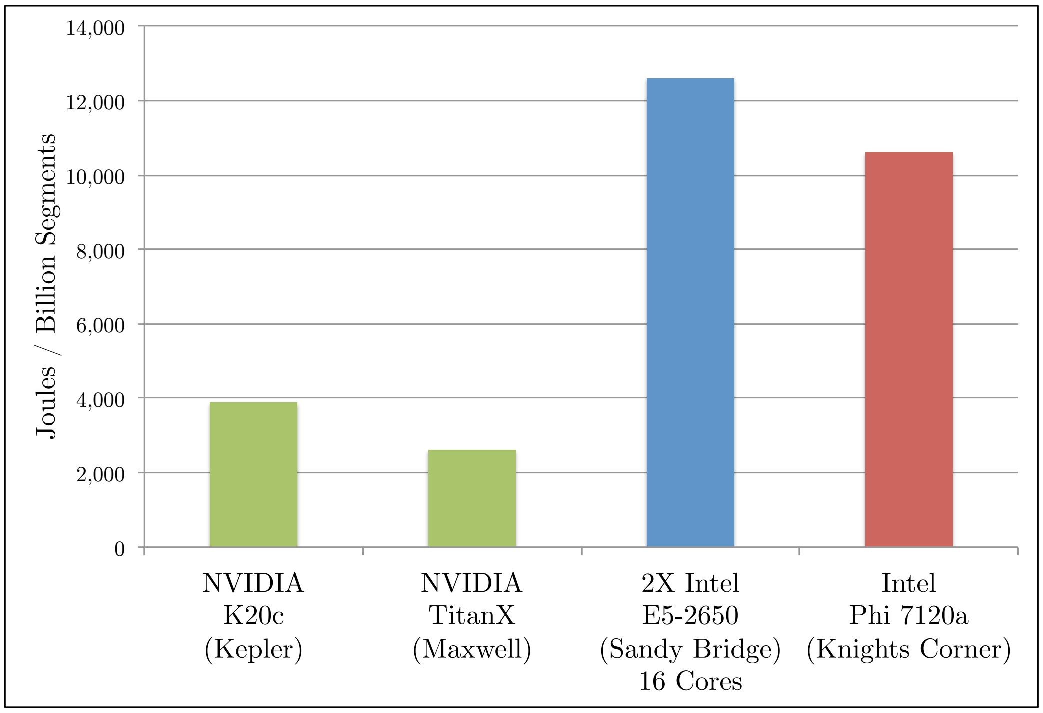


Figure 9: SimpleMOC-kernel total energy (Joules) to process 1 billion segments for various architectures (lower is better).

modern computer architectures to be fully utilized. With our representation of the 3D MOC algorithm, we were able to achieve 63% of peak FLOPS on a dual socket Intel E5-2650 Sandy Bridge node. Additionally, we identified the remaining bottlenecks of the algorithm on all three architectures, namely, register pressure on CPU, memory latency on GPU, and insufficient cache on the Intel Phi.

Ultimately, we found that the GPU platform delivered superior performance in both speed, energy efficiency, and cost efficiency when compared to CPU or Phi nodes operating with all cores and threads. Impressively, the NVIDIA TitanX GPU outperformed the dual socket Intel E5-2699v3 Haswell 36-core CPU node by a factor of 2.3x, while simultaneously improving cost efficiency by a factor of over 23x. These findings suggest that the 3D MOC algorithm is well suited to achieving high performance on the GPU platform, making the CUDA language an excellent candidate for implementation of a full scale exascale class 3D MOC solver for reactor simulation applications.

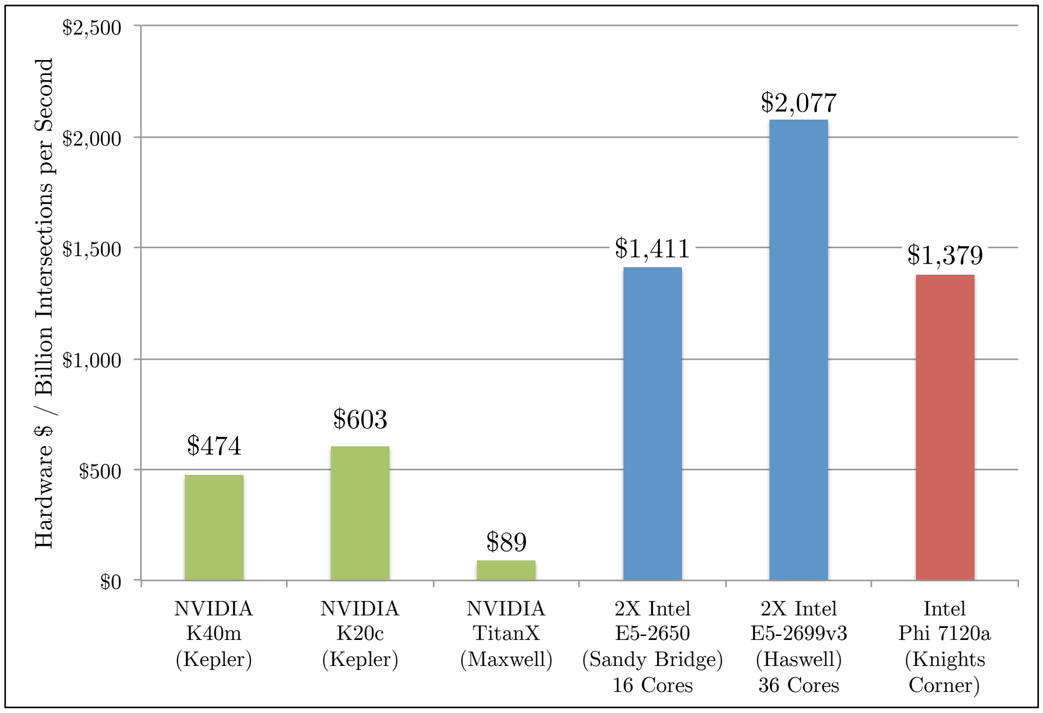


Figure 10: SimpleMOC-kernel - Hardware cost ($) per 3D MOC performance unit (billion intersections/sec)

Additionally, the next generation NVIDIA "Pascal" GPU architecture (along with many other low power architectures) are slated to feature mixed and half precision (FP16) hardware support, which may prove advantageous for our implementation of the 3D MOC algorithm.

It is critical to note that these findings form an important stepping stone on the road towards the development of a full scale production 3D MOC application. This in-depth performance analysis provides a foundational understanding of 3D MOC reactor simulation on cutting edge high performance computing architectures and offers a road map for selection of future algorithmic choices, programming languages, and architecture targets for exascale reactor simulations.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract

DE-AC02-06CH11357. The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract DE-AC02-06CH11357 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

The second author is a recipient of the U.S. Department of Energy Office of Nuclear Energy's Nuclear Energy University Programs Fellowship. Additional support was provided by the Center for Exascale Simulation of Advanced Reactors (CESAR), a co-design center under the U.S. Department of Energy. The authors would also like to thank Michael Smith from Argonne National Laboratory for providing guidance from his experience with 3D MOC solvers.

We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

References

[1] B. Kochunas, A hybrid parallel algorithm for the 3-D Method of Characteristics solution of the Boltzmann Transport Equation on high performance computing clusters, Ph.D. Thesis, University of Michigan, Department of Nuclear Engineering and Radiological Sciences (2013).

[2] G. Palmiotti, M. Smith, C. Rabiti, M. Leclere, D. Kaushik, A. Siegel, B. Smith, E. Lewis, UNÌC: Ultimate neutronic investigation code, in: M&C + SNA 2007 – Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications, Monterey, California, 2007.

[3] M. A. Smith, D. Kaushik, A. Wollaber, W. S. Yang, B. Smith, C. Rabiti, G. Palmiotti, Recent research progress on UNÌC at Argonne National Laboratory, in: International Conference on Mathematics, Computational Methods & Reactor Physics, Saratoga Springs, New York, 2009.

[4] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, Exascale computing study: Technology challenges in achieving exascale systems, http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf (2008).

[5] G. Gunow, J. R. Tramm, B. Forget, K. Smith, SimpleMOC – A performance abstraction for 3D MOC, in: ANS&MC 2015 – Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method, Nashville, 2015.

[6] W. Boyd, Massively parallel algorithms for Method of Characteristics neutral particle transport on shared memory computer architectures, M.S. Thesis, Massachusetts Institute of Technology, Department of Nuclear Science and Engineering (2014).

[7] R. Sanchez, Prospects in deterministic three-dimensional whole-core transport calculations, Nuclear Engineering and Technology 44 (2) (2012) 113–150.

[8] S. Shaner, et al., Theoretical Analysis of Track Generation in 3D Method of Characteristics, *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering* (2015).

[9] N. Horelik, B. Herman, B. Forget, K. Smith, Benchmark for evaluation and validation of reactor simulations (BEAVRS), in: M&C 2013 – International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering, 2013.

[10] J. Tramm, G. Gunow, SimpleMOC: A 3D Method of Characteristics (MOC) reactor simulation mini-app featuring MPI and OpenMP parallelism, https://github.com/ANL-CESAR/SimpleMOC (2015).

[11] R. Rahaman, D. Medina, A. Lund, J. Tramm, T. Warburton, A. Siegel, Portability and performance of nuclear reactor simulations on many-core architectures, in: EASC 2015 – Exascale Applications and Software Conference, 2015.

[12] J. Tramm, G. Gunow, SimpleMOC-kernel: A 3D Method of Characteristics (MOC) reactor simulation kernel mini-app, https://github.com/ANL-CESAR/SimpleMOC-kernel (2015).

[13] Innovative Computing Laboratory, PAPI - Performance application programming interface, http://icl.cs.utk.edu/papi/index.html (2015).

[14] J. Dongarra, P. Luszczek, LINPACK benchmark, in: Encyclopedia of Parallel Computing, Springer US, 2011, pp. 1033–1036. doi:10.1007/978-0-387-09766-4_155.